C++ Coroutines and Qt

Meeting C++ 2021

Daniel Vrátil

me@dvratil.cz

@danvratil | danvratil

Contents

- Quick Qt introduction
- The problem
- The solution
- Conclusion

Contents

Quick Qt Introduction - Qt Event Loop

- event loop like any other
- one per thread
- don’t nest them

Quick Qt Introduction - Signals & Slots

- Signals
  - regular member functions
  - only declared
  - definition provided by MOC
  - when “emitted” (called), Qt invokes all slots connected the signal
  - multiple slots can be connected to a single signal
- Slots
  - regular functions
  - invoked by Qt whenever a signal to which slot is connected is emitted
  - single slot can be connected to multiple signals

Contents

- Quick Qt introduction
- The problem
  - Async operations with Qt
- The solution
- Conclusion

The Problem - Async Operations

```
QString APIService::getUserName(uint64_t userId) {
    QNetworkAccessManager nam;
    auto *reply = nam.get(…);

    reply→waitForReadyRead();

    const QString userName = QString::fromUtf8(reply→readAll());
    reply→deleteLater();
    return userName;
}
```

(please don’t do this)

The Problem - Async Operations

```
QString APIService::getUserName(uint64_t userId) {
    QNetworkAccessManager nam;
    auto *reply = nam.get(…);

    reply→waitForReadyRead();

    const QString userName = QString::fromUtf8(reply→readAll());
    reply→deleteLater();
    return userName;
}
```

(please don’t do this)

The Problem - Async Operations

```
QString APIService::getUserName(uint64_t userId) {
    QNetworkAccessManager nam;
    auto *reply = nam.get(…);

    reply→waitForReadyRead();

    const QString userName = QString::fromUtf8(reply→readAll());
    reply→deleteLater();
    return userName;
}
```

(please don’t do this)

The Problem - Async Operations

```
QString APIService::getUserName(uint64_t userId) {
    QNetworkAccessManager nam;
    auto *reply = nam.get(…);

    reply→waitForReadyRead();

    const QString userName = QString::fromUtf8(reply→readAll());
    reply→deleteLater();
    return userName;
}
```

(please don’t do this)

The Problem - Async Operations

```
void APIService::getUserName(uint64_t userId, Callback cb) {
    QNetworkAccessManager nam;
    auto *reply = nam.get(…);

    connect(reply, &QNetworkReply::finished,
            this, [=]() {
                cb(QString::fromUtf8(reply→readAll()));
                reply→deleteLater();
            });
}
```

The Problem - Async Operations

```
const auto userName = apiService.getUserName(userId);
doSomethingWithUserName(userId, userName);
```

```
apiService.getUserName(userId, [=](const auto &userName) {
    doSomethingWithUserName(userId, userName);
});
```

The Problem - Async Operations

```
const auto userName = apiService.getUserName(userId);
doSomethingWithUserName(userId, userName);
```

```
apiService.getUserName(userId, [=](const auto &userName) {
    doSomethingWithUserName(userId, userName);
});
```

Summary - Problems with async (in Qt)

- Chaining asynchronous operations is hard
- Error propagation is hard
- The code is hard to follow (especially with signals & slots where slots are implemented elsewhere)

Contents

The Solution - Coroutines

```
QCoro::Task<QString> APIService::getUserName(uint64_t userId) {
    QNetworkAccessManager nam;
    auto *reply = co_await nam.get(…);

    const QString userName = QString::fromUtf8(reply→readAll());
    reply→deleteLater();
    co_return userName;
}
```

The Solution - Coroutines (cont.)

- co_await suspends the coroutine
- execution returns to the caller - (which is likely co_awaiting our coroutine, so it suspends as well)
- this repeats all the way until we reach the Qt event loop
- the event loop dispatches the network request
- when the request finishes, it resumes the coroutine
- the coroutine provides a result to the caller, which gets resumed as well
- this repeats until the original callstack is all done and back to event loop

The Solution - Coroutines (cont.)

- co_await suspends the coroutine
- execution returns to the caller - (which is likely co_awaiting our coroutine, so it suspends as well)
- this repeats all the way until we reach the Qt event loop
- the event loop dispatches the network request
- when the request finishes, it resumes the coroutine
- the coroutine provides a result to the caller, which gets resumed as well
- this repeats until the original callstack is all done and back to event loop

The Solution - Coroutines (cont.)

- `QCoro::Task`
  - a return type that can be `co_await`ed by callers
- `QCoro::TaskPromise`
  - a promise type provided by the `QCoro::Task` type
  - implements `await_transform(T &&value)` for any type T for which `QCoro::detail::awaiter_type<T>` trait is defined
  - the trait provides wrapper type for type T, which can be `co_await`ed

The Solution - Coroutines

```
namespace QCoro::detail {
template<>
struct awaiter_type<QNetworkReply *> {
    using type = QCoro::detail::QCoroNetworkReply::WaitForFinishedOperation;
};
}
```

The Solution - Coroutines

```
namespace QCoro::detail {
class QCoroNetworkReply : … {
    …
    class WaitForFinishedOperation {
        explicit WaitForFinishedOperation(QNetworkReply *);
        bool await_ready() const noexcept;
        void await_suspend(std::coroutine_handle<> awaitingCoro);
        QNetworkReply *await_resume() const noexcept;
    };
    …
};
}
```

The Solution - Coroutines

```
namespace QCoro::detail {
class QCoroNetworkReply : … {
    …
    class WaitForFinishedOperation {
        explicit WaitForFinishedOperation(QNetworkReply *);
        bool await_ready() const noexcept;
        void await_suspend(std::coroutine_handle<> awaitingCoro);
        QNetworkReply *await_resume() const noexcept;
    };
    …
};
}
```

The Solution - Coroutines

```
namespace QCoro::detail {
class QCoroNetworkReply : … {
    …
    class WaitForFinishedOperation {
        explicit WaitForFinishedOperation(QNetworkReply *);
        bool await_ready() const noexcept;
        void await_suspend(std::coroutine_handle<> awaitingCoro);
        QNetworkReply *await_resume() const noexcept;
    };
    …
};
}
```

The Solution - Coroutines

```
namespace QCoro::detail {
class QCoroNetworkReply : … {
    …
    class WaitForFinishedOperation {
        explicit WaitForFinishedOperation(QNetworkReply *);
        bool await_ready() const noexcept;
        void await_suspend(std::coroutine_handle<> awaitingCoro);
        QNetworkReply *await_resume() const noexcept;
    };
    …
};
}
```

The Solution - Coroutines

```
namespace QCoro::detail {
class QCoroNetworkReply : … {
    …
    class WaitForFinishedOperation {
        explicit WaitForFinishedOperation(QNetworkReply *);
        bool await_ready() const noexcept;
        void await_suspend(std::coroutine_handle<> awaitingCoro);
        QNetworkReply *await_resume() const noexcept;
    };
    …
};
}
```

The Solution - QCoro

- MIT-licensed library
- GCC, Clang, MSVC
- provides coroutine-friendly wrappers for various standard Qt classes
- provides QCoro::Task to automatically wrap supported Qt class into a coroutine-friendly wrapper
- QIODevice, QProcess, QTimer, QFuture, QAbstractSocket, QLocalSocket, QNetworkReply, QTcpServer, QDBusPendingCall, QDBusReply
- awaiting signal emissions

The Solution - QCoro

- Some types may have multiple operations we want to await asynchronously:
- `QProcess::waitForStarted()`, `QProcess::waitForFinished()`
- can't just do `co_await process` - which operation should be awaited?
- `qCoro()` wraps QProcess into a QCoroProcess, which has awaitable versions of those methods:

```
const bool started = co_await qCoro(process).waitForStarted();
...
const bool finished = ci_await qCoro(process).waitForFinished();
```

The Solution - QCoro - awaiting signals

```
QProcess process;
process.start(“/usr/bin/gpg2”, {“-d”});

co_await qCoro(process).waitForStarted();

process.write(encryptedData);
process.closeWriteChannel();

co_await qCoro(process).waitForFinished();
const QByteArray decryptedData = process.readAll();
```

The Solution - QCoro - awaiting signals

```
QProcess process;
process.start(“/usr/bin/gpg2”, {“-d”});

co_await qCoro(process).waitForStarted();

process.write(encryptedData);
process.closeWriteChannel();

co_await qCoro(process).waitForFinished();
const QByteArray decryptedData = process.readAll();
```

The Solution - QCoro - awaiting signals

```
QProcess process;
process.start(“/usr/bin/gpg2”, {“-d”});

co_await qCoro(process).waitForStarted();

process.write(encryptedData);
process.closeWriteChannel();

co_await qCoro(process).waitForFinished();
const QByteArray decryptedData = process.readAll();
```

The Solution - QCoro - awaiting signals

```
QProcess process;
process.start(“/usr/bin/gpg2”, {“-d”});

co_await qCoro(process).waitForStarted();

process.write(encryptedData);
process.closeWriteChannel();

co_await qCoro(process).waitForFinished();
const QByteArray decryptedData = process.readAll();
```

The Solution - QCoro - awaiting signals

```
class ProcessMonitor: public QObject {
   Q_OBJECT
    …
Q_SIGNALS:
    void processStarted(const QString &name);
};


ProcessMonitor monitor;
const auto name =
    co_await qCoro(&monitor, &ProcessMonitor::processStarted);
```

The Solution - QCoro - awaiting signals

```
class ProcessMonitor: public QObject {
   Q_OBJECT
    …
Q_SIGNALS:
    void processStarted(const QString &name);
};


ProcessMonitor monitor;
const auto name =
    co_await qCoro(&monitor, &ProcessMonitor::processStarted);
```

The Solution - QCoro - awaiting signals

```
class ProcessMonitor: public QObject {
   Q_OBJECT
    …
Q_SIGNALS:
    void processStarted(const QString &name);
};


ProcessMonitor monitor;
const auto name =
    co_await qCoro(&monitor, &ProcessMonitor::processStarted);
```

The Solution - QCoro - awaiting signals

```
class ProcessMonitor: public QObject {
   Q_OBJECT
    …
Q_SIGNALS:
    void processFinished(const QString &name, int result);
};


ProcessMonitor monitor;
const auto [name, result] =
    co_await qCoro(&monitor, &ProcessMonitor::processFinished);
```

The Solution - QCoro - simple server

```
QTcpServer server;
server.listen();
while (true) {
   auto *conn = co_await qCoro(server).waitForNewConnection();
   const auto data = co_await qCoro(conn)→readAll();
   conn→write(data);
   conn→close();
}
```

The Solution - QCoro - simple server

```
QTcpServer server;
server.listen();
while (true) {
    auto *conn = co_await qCoro(server).waitForNewConnection();
    const auto data = co_await qCoro(conn)→readAll();
    conn→write(data);
    conn→close();
}
```

The Solution - QCoro - simple server

```
QTcpServer server;
server.listen();
while (true) {
    auto *conn = co_await qCoro(server).waitForNewConnection();
    const auto data = co_await qCoro(conn)→readAll();
    conn→write(data);
    conn→close();
}
```

The Solution - QCoro - simple server

```
QTcpServer server;
server.listen();
while (true) {
    auto *conn = co_await qCoro(server).waitForNewConnection();
    const auto data = co_await qCoro(conn)→readAll();
    conn→write(data);
    conn→close();
}
```

The Solution - QCoro - simple server

```
QTcpServer server;
server.listen();
while (true) {
    auto *conn = co_await qCoro(server).waitForNewConnection();
    const auto data = co_await qCoro(conn)→readAll();
    conn→write(data);
    conn→close();
}
```

The Solution - QCoro - QtConcurrent support

```
const int theAnswer = co_await QtConcurrent::run([]() {
    std::this_thread::sleep_for(std::chrono::years{7’500’000});
    return 42;
});
```

The Solution - QCoro - Future?

- Support for certain GUI/widget classes
  - e.g. awaiting dialogs
- `finished()` signal in the `QCoro::Task` class
- truly asynchronous file and directory operations
- ideas and contributions welcomed ;-)

Questions?

QCoro
github.com/danvratil/qcoro
qcoro.dvratil.cz

Daniel Vrátil
me@dvratil.cz
@danvratil | danvratil